Wavelet Trees

02-714
Slides by Carl Kingsford

Following: Navarro,
Wavelet Trees for All, CPM 2012, pp. 2-26
Operations on strings

- \( \text{rank}_c(S, i) := \) the number of char \( c \) at or before position \( i \) in \( S \).
- \( \text{select}_c(S, j) := \) the position of the \( j^{\text{th}} \) occurrence of \( c \) in \( S \).
- \( S[i] = \) “access character \( i \)”

Note: \( \text{rank}_c(S, \text{select}_c(S, j)) = j \), so rank and select are inverses of each other.

**Goal:** rank, select, access in quickly while using small space.
Operations on bit vectors

Our operations will depend on similar operations on bit vectors:

- \( \text{rank}_1(S, i) := \) the number of 1 bits at or before position \( i \) in \( S \).
- \( \text{select}_1(S, j) := \) the position of the \( j^{\text{th}} \) 1 bit in \( S \).
- \( \text{rank}_0(S, i) \) and \( \text{select}_0(S, j) \) are defined analogously.

\[ S[i] = \text{“access bit } i\text{” } = \text{rank}_1(S, i) - \text{rank}_1(S, i-1) \]

We will see later how to implement these operations to run in \( O(1) \) time for binary vectors.
Wavelet Tree

\[ S = A C G G G A C C G T T T T T A G G A \]

0 0 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 0

0: letter \( \in \) first half of alphabet
1: letter \( \in \) first half of alphabet
Wavelet Tree, Example 2
**S[i]**

Go left if bit is 0, go right if bit is 1.

Use $\text{rank}_01()$ to map a bit at a node to the right place in the children.
**S[i]**

Go left if bit is 0, go right if bit is 1.

Use \( \text{rank}_{01}(\) to map a bit at a node to the right place in the children.
Go left if bit is 0, go right if bit is 1.

Use rank$_{01}(\cdot)$ to map a bit at a node to the right place in the children.
$S[i]$

Go left if bit is 0, go right if bit is 1.

Use $\text{rank}_{01}()$ to map a bit at a node to the right place in the children.

- $i_v = \text{rank}_0(\text{root}, i)$
- $i_w = \text{rank}_1(v, i_v)$
- $i_3 = \text{rank}_0(w, i_w) = 5$
Use $\text{rank}(i)$ to map a bit at a node to the right place in the children.

Go left if $c$ is in first half of alphabet, go right if $c$ is in second half of alphabet.
**rank**\(_c(S,i)\)**

Go left if \(c\) is in **first half of alphabet**, go right if \(c\) is in second half of alphabet.

Use \(\text{rank}_{01}()\) to map a bit at a node to the right place in the children.

![Diagram of rank calculation](image)
\( \text{rank}_c(S,i) \)

Go left if \( c \) is in first half of alphabet, go right if \( c \) is in second half of alphabet.

Use \( \text{rank}_{01}() \) to map a bit at a node to the right place in the children.
**rank\textsubscript{c}(S,i)**

Go left if **c** is in **first half of alphabet**, go right if **c** is in second half of alphabet.

Use \text{rank}_{01()} to map a bit at a node to the right place in the children.
**select\textsubscript{c}(S,j)**

Start at position \( j \) in leaf corresponding to \( c \).

Repeat: new \( i = \text{select}_b(p, i) \) where \( p = \) parent of current node, \( b = 0 \) if current node is left child of \( p \), \( 1 \) if right child.

---

![Diagram showing the process of select\textsubscript{c}(S,j) with a binary search tree and the path from root to the leaf at position j=4.](image)
**select\textsubscript{c}(S, j)**

Start at position \( j \) in leaf corresponding to \( c \).

Repeat: new \( i = \text{select}_b(p, i) \) where \( p = \) parent of current node, \( b = 0 \) if current node is left child of \( p \), \( 1 \) if right child.

\[ \text{select}_0(w, 4) = i_w \]

\[ j = 4 \]

\[ \text{select}_3(S, 4) \]
select\(_c(S,j)\)

Start at position \(j\) in leaf corresponding to \(c\).

Repeat: new \(i = \text{select}_b(p, i)\) where \(p = \text{parent of current node}\), \(b = 0\) if current node is left child of \(p\), 1 if right child.
select\textsubscript{c}(S, j)

Start at position \( j \) in leaf corresponding to \( c \).

Repeat: new \( i = \text{select}_{b}(p, i) \) where \( p = \) parent of current node, \( b = 0 \) if current node is left child of \( p \), 1 if right child

\( i_{\text{root}} = \text{select}_{0}(\text{root, } i_{v}) \)

\( i_{v} = \text{select}_{1}(w, i_{w}) \)

\( i_{w} = \text{select}_{0}(w, 4) \)

\( j=4 \)

select\textsubscript{3}(S, 4)
**Running Times**

Tree height = $\log |\Sigma|$, where $\Sigma$ is the alphabet.

rank, select, access follow a root-leaf path in the tree, taking $O(1)$ time at each node.

Therefore, rank, select, access take $O(\log |\Sigma|)$ to run.

If alphabet size is constant, this is $O(1)$. 
Tree Shape

- Don’t have to use balanced tree shape
- Can instead encode using, say, Huffman code tree shape
  - makes accesses to frequent characters faster
  - gives good space usage even without compressed bit vectors.
- Doesn’t have to be binary tree
  - by selecting optimal branching factor, can get query time to $O(\log n / \log \log n)$. 
Application: Inverted Indices

If you really want to hear about it, the first thing you'll probably want to know is where I was born, and what my lousy childhood was like, and how my parents were occupied and all before they had me, and all that David Copperfield kind of crap, but I don't feel like going into it, if you want to know the truth.

Traditional inverted index represents the document as an array of lists:

<table>
<thead>
<tr>
<th>If</th>
<th>1, 57</th>
</tr>
</thead>
<tbody>
<tr>
<td>you</td>
<td>2, 58</td>
</tr>
<tr>
<td>really</td>
<td>3</td>
</tr>
<tr>
<td>want</td>
<td>4, 14, 59</td>
</tr>
<tr>
<td>and</td>
<td>22, 29, 35, 41</td>
</tr>
<tr>
<td>know</td>
<td>16, 61</td>
</tr>
<tr>
<td>truth</td>
<td>63</td>
</tr>
</tbody>
</table>

Good for searching for word $w$

Hard to compute $S[i]$
Application: Inverted Indices, 2

- Represent text as a string of word ids.

- Store as a wavelet tree.

- $S[i]$ now $O(\log |\Sigma|) = O(\log n)$

- $\text{select}_w(S, j)$ now gives the position of the jth occurrence of word $w$ in time $O(\log n)$. 
Application: Document Retrieval

Given a collection of documents $D_1, \ldots, D_m$, answer the following types of queries quickly:

- In which documents does word $w$ appear?

Represent documents as strings of word ids.

Concatenate the documents together, separated by a $\$ \$ \$ \$ word that does not occur elsewhere.

\[
i = 1
\]

\textbf{repeat:}

\[
\begin{align*}
  p_i &= \text{select}_c(i) & \# \text{ith occurrence} \\
  d_i &= \text{rank}_\$ (p_i) + 1 & \# \text{document containing it} \\
  \text{print } d_i \\
  p' &= \text{select}_\$(d_i) & \# \text{end of document} \\
  i &= \text{rank}_c(p') + 1 & \# \text{find 1st occurrence after end of doc}
\end{align*}
\]
Application: Graphs

Given a directed graph G, answer the following queries quickly:

- successor(u, i) := the ith vertex v such that edge (v, u) exists.
- predecessor(u, i) := the jth vertex v such that edge (u, v) exists.

Represent G as a concatenation of adjacencies lists, and store in wavelet tree:

\begin{align*}
\text{bit vector} & : \quad 2 \ 3 \ 5 \ 4 \ 3 \ 5 \ 6 \ 2 \ 6 \ 5 \ 4 \ 1 \ 2 \\
\text{marking list} & : \quad 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\
\end{align*}

- successor(u, i): \( p = \text{select}_1(B, u) \); return \( S[p+i-1] \)
- predecessor(u, i): \( p = \text{select}_u(S, i) \); return \( \text{rank}_1(B, p) \)

\( |\Sigma| = n \), so these take \( O(\log n) \) time.
Application: Grid of Points

Suppose you have points \((x_1, y_1), \ldots, (x_n, y_n)\) on an \(m \times m\) grid and you want to answer range queries quickly:

- Which points fall inside rectangle \([x_{\text{min}}, x_{\text{max}}]\) by \([y_{\text{min}}, y_{\text{max}}]\)?

Sort points by x-coordinate, consider \(y_{\pi(1)}, y_{\pi(4)}, y_{\pi(3)}, \ldots, y_{\pi(n)}\) as a string and store in a wavelet tree.

Map range to left and right, stop when characters in interval are contained in \([y_{\text{min}}, y_{\text{max}}]\).

\[
x'_{\text{left}} = \text{rank}_0(B_{\text{root}}, x_{\text{left}})
\]

\[
x'_{\text{right}} = \text{rank}_0(B_{\text{root}}, x_{\text{right}})
\]

\[
x'_{\text{left}} = \text{rank}_1(B_{\text{root}}, x_{\text{left}})
\]

\[
x'_{\text{right}} = \text{rank}_1(B_{\text{root}}, x_{\text{right}})
\]
Summary

- Wavelet trees compactly store strings.
- Allowing access almost as fast as for a plain array.
- And allowing for fast rank and select queries too.
- Lots of applications and extensions, often storing things not normally thought of as strings.